Spud 1.0.9 Manual

February 27, 2009

Contents

Chapter 1

Installation and building

1.1 Dependencies

Spud and Diamond depend on the following packages:

- Fortran, C and C++ compilers
- Python (`http://python.org/`)
- Python setuptools (`http://peak.telecommunity.com/DevCenter/setuptools`)
- PyGTK (`http://www.pygtk.org/`)
- lxml (`http://codespeak.net/lxml/`)
- Trang (`http://www.thaiopensource.com/relaxng/trang.html`)
- libxml2 (`http://xmlsoft.org/`)
- LaTeX (for the manual) (`http://www.latex-project.org/`)

Users of Debian and Ubuntu should be able to install all the needed dependencies by adding the repository lines listed below and typing:

```
sudo apt-get update
sudo apt-get build-depend spud
```

1.2 Obtaining the source

Spud is managed using Subversion. Version 1.0.9 can be obtained with the command:

```
svn co http://amcg.ese.ic.ac.uk/svn/spud/branches/1.0.9 spud
```

the current development version is available via:

```
svn co http://amcg.ese.ic.ac.uk/svn/spud/trunk spud
```

Debian and Ubuntu users can obtain the source package by adding the repository lines listed below and typing:

```
sudo apt-get update
sudo apt-get source spud
```

1.3 Building from source

Spud is built using a standard autoconf system. It should be possible to build spud and install it in `/usr/local` simply by typing:

```
./configure
make
make install
```

Installing to an alternative location is possible by specifying the `--prefix` option to `configure`. For a full list of configure options type:

```
./configure --help
```

1.4 Debian and Ubuntu packages

Debian and Ubuntu packages are available from the Applied Modelling and Computation Group at Imperial College London. The Debian repository may be accessed by adding the following lines to `/etc/apt/sources.list`:

```
deb http://amcg.ese.ic.ac.uk/debian/ unstable main contrib non-free
deb-src http://amcg.ese.ic.ac.uk/debian/ unstable main contrib non-free
```

The Ubuntu repository may be accessed via the following lines:

```
deb http://amcg.ese.ic.ac.uk/debian/ gutsy main contrib non-free
deb-src http://amcg.ese.ic.ac.uk/debian/ gutsy main contrib non-free
```

or:

```
deb http://amcg.ese.ic.ac.uk/debian/ hardy main contrib non-free
deb-src http://amcg.ese.ic.ac.uk/debian/ hardy main contrib non-free
```

The Spud library and the base language are installed by the `libspud-dev` package while diamond is shipped in the `diamond` package. Both packages are built from the `spud` source package. Binary packages are supplied for the i386 and amd64 architectures.

After adding the relevant lines to `/etc/apt/sources.list`, install the packages by typing:

```
sudo apt-get update
sudo apt-get install diamond libspud-dev
```

Chapter 2

Schemas and the Spud base language

The World Wide Web Consortium's Extensible Markup Language (XML) provides a generic syntax for machine parseable languages. These allow the organisation of model input options into a tree of nested elements. Utilising such a structure within the options file allows distinct groups of options to be gathered together in branches while suboption dependencies can be represented as child elements.

2.1 RELAX NG

Spud uses the RELAX NG schema language within the XML system. For full documentation of RELAX NG see `http://relaxng.org/`. The compact syntax tutorial is particularly useful.

The examples presented here are shown in a compact syntax of RELAX NG. This is the preferred syntax for editing Spud schemas and the format it is shipped in. However the more verbose XML syntax is better supported by software parsers. Hence the completed schema, including the base language, is translated from compact to XML syntax using the software package Trang before use by Spud based tools like Diamond.

2.2 Base language named patterns

The RELAX NG language allows different schemas to be imported into one another, which enables Spud to define a base language for schema developers. The Spud base language thus provides core schema objects (known in RELAX NG as patterns) that enable generic tools included in Spud to handle low level data in an elegant manner.

For example the `real_dim_symmetric_tensor` pattern is defined below:

```
# A dim x dim real matrix (rank 2 tensor) constrained to be symmetric.
real_dim_symmetric_tensor =
   (
      element real_value{
         attribute symmetric {"true"},
         attribute rank { "2" },
         # Setting dim1, dim2 to a function of dim allows the gui
         # to set the tensor to the right shape.
         attribute dim1 { "dim" },
         attribute dim2 { "dim" },
         attribute shape { list{xsd:integer, xsd:integer} },
         list {xsd:float+}
      },
      comment
```

```
   )
```

This core object contains all the information required to define the properties of a real, symmetric, rank 2 tensor with square dimensions equal to the physical dimension specified. This enables Spud based generic tools to reduce the level of information required as input from the user. In this case the user is only required to provide a list of reals to fill out the tensor while the generic tool will ensure it is symmetric, ordered correctly and save the rank, shape and dimensions. Thus from the developers perspective all the information required to import a symmetric tensor is available from Spud alongside the user's input.

As can be seen above the principal element of the `real_dim_symmetric_tensor` pattern is `real_value`. Modification of the rank, shape and dimension attributes of this element allows the Spud based language to be expanded easily to incorporate other real data structures. For instance, a rank 1 real vector with length equal to the physical dimension specified is defined in the `real_dim_vector` pattern:

```
# A real vector of length dim
real_dim_vector =
   (
      element real_value{
         attribute rank { "1" },
         # Setting dim1 to a function of dim allows the gui to set the
         # vector to the right length.
         attribute dim1 { "dim" },
         attribute shape { xsd:integer },
         list{xsd:float+}
      },
      comment
   )
```

Similar extensions can be made for an integer based `integer_value` element, while a `string_value` element allows the definition of several character patterns known to Spud generic tools (such as the `comment` pattern seen in the examples above). This allows for the definition of the full Spud base language as follows:

`comment`	A string value that allows users to annotate their input throughout the XML tree structure. Included in all Spud base language patterns.
`anystring`	A string value for the input of any generic character string required in the options tree. Suggests a display of 1 line within Spud based tools such as Diamond.
`filename`	A string value for the input of filenames in the options tree. Allows the use of a file selector and suggests a display of 1 line within Spud based tools such as Diamond.
`python_code`	A string value for the input of python code into the options tree. Suggests a display of 20 lines within Spud based tools such as Diamond.
`integer`	An integer value of rank 0 and length 1.
`integer_vector`	A rank 1 vector of integers of arbitrary length. Spud tools record the shape of the input.
`integer_tensor`	A tensor of integers of arbitrary dimensions. Spud tools record the shape of the input.
`integer_dim_vector`	A rank 1 vector of integers with length equal to the physical dimension specified.

integer_dim_minus_one_vector	A rank 1 vector of integers with length equal to 1 less than the physical dimension specified.
integer_dim_tensor	A rank 2 tensor of integers with square dimensions equal to the physical dimension specified.
integer_dim_symmetric_tensor	A rank 2 symmetric tensor of integers with square dimensions equal to the physical dimension specified.
integer_dim_minus_one_tensor	A rank 2 tensor of integers with square dimensions equal to 1 less than the physical dimension specified.
integer_dim_minus_one_symmetric_tensor	A rank 2 symmetric tensor of integers with square dimensions equal to 1 less than the physical dimension specified.
real	A real value of rank 0 and length 1.
real_vector	A rank 1 vector of reals of arbitrary length. Spud tools record the shape of the input.
real_tensor	A rank 2 tensor of reals of arbitrary dimensions. Spud tools record the shape of the input.
real_dim_vector	A rank 1 vector of reals with length equal to the physical dimension specified.
real_dim_minus_one_vector	A rank 1 vector of reals with length equal to 1 less than the physical dimension specified.
real_dim_tensor	A rank 2 tensor of reals with square dimensions equal to the physical dimension specified.
real_dim_symmetric_tensor	A rank 2 tensor of reals with square dimensions equal to the physical dimension specified.
real_dim_minus_one_tensor	A rank 2 tensor of reals with square dimensions equal to 1 less than the physical dimension specified.
real_dim_minus_one_symmetric_tensor	A rank 2 tensor of reals with square dimensions equal to 1 less than the physical dimension specified.

2.2.1 Including the base language in the schema

The base language is described in the file `spud_base.rnc`. This file is installed in `@prefix@/share/spud`, where the default value of `@prefix@` is `/usr`. It may be referenced without prefixing the path. The top line of every spud schema should read

```
include "spud_base.rnc"
```

2.2.2 The real_value and integer_value elements

As mentioned above, the `real_value` and `integer_value` elements are the low-level implementations of the real and integer named patterns in the base language. The full schema syntax of these elements is illustrated by the definition of the `real_dim_minus_one_tensor` pattern in the base language:

```
# A dim-1 x dim-1 real matrix (rank 2 tensor).
real_dim_minus_one_tensor =
   (
      element real_value{
         attribute symmetric {"false"},
```

```
            attribute rank { "2" },
            # Setting dim1, dim2 to a function of dim allows the gui to set the
            # tensor to the right shape.
            attribute dim1 { "dim-1" },
            attribute dim2 { "dim-1" },
            attribute shape { list{xsd:integer, xsd:integer} },
            list {xsd:float+}
        },
        comment
    )
```

The `symmetric` and `dim2` attributes are only present if `rank` is 2 while the `dim1` and `shape` attributes are only present where `rank` is at least 1. If `rank` is equal to 1 then `shape` will be a single `xsd:integer` rather than a list of 2. The `dim1` and `dim2` attributes are python expressions of the variable `dim`.

2.2.3 The string_value element

Customised string values, such as fixed strings or multiline strings, are constructed using the `string_value` element. This element wraps a string and a `lines` attribute. The `lines` attribute does not enforce a length on the string but is rather a hint to the user interface as to the size of string box which would be appropriate. The schema syntax of the `string_value` element is illustrated by the definition of the main `anystring pattern` from the base language:

```
# A simple string
anystring =
    (
        element string_value{
            # Lines is a hint to the gui about the size of the text box.
            # It is not an enforced limit on string length.
            attribute lines { "1" },
            xsd:string
        },
        comment
    )
```

An example of a customised string is this choice of two available string values:

```
## Format for dump files. Choose from fluidity dumpfile or vtk.
element dump_format {
    element string_value{
        "fluidity␣dumpfile"|"vtk"
    }
}
```

2.2.4 Symmetric tensors

Some of the Spud base language patterns refer to symmetric tensors. In every case, the full tensor is stored and retrieving the tensor using libspud. It is the responsibility of Spud user interface tools such as Diamond to enforce the symmetry of tensors input by the user.

```
include "spud_base.rnc"

start =
   (
      # Outside wrapper element. Doesn't really matter what the name is.
      element model_options {
          comment,
         ## Model output files are named according to the simulation name,
         ## e.g. [simulation_name]_0.vtu. Non-standard characters in the
         ## simulation name should be avoided.
         element simulation_name {
            anystring
         },
         ## Options dealing with the specification of geometry
         element geometry {
            ## Dimension of the problem.
            ## <b>This can only be set once</b>
            element dimension {
               attribute replaces {"NDIM"},
               element integer_value {
                  attribute rank {"0"},
                  ("3"|"2")
               }
            }
         }
      }
   )
```

Figure 2.1: A trivial schema showing a schema comment, schema annotations and a user comment pattern. The dimension of the problem is specified by the dimension element under the geometry element.

2.3 Specifying problem dimension

The dimension of the problem affects how all of the dimension-specific named patterns in the schema are handled. It is specified by having an integer-valued `dimension` element as a child of the `geometry` element which is in turn a child of the root element. Spud schemas are required to define `dimension` and `geometry` elements if they make use of any of the named patterns whose name includes the string `dim`. Figure 2.1 illustrates a trivial Spud schema in which the dimension attribute is given using a customised `integer_value` element which only permits the values 2 or 3 to be given.

2.4 Restrictions on the base language

In order to facilitate processing of schemas by libspud and Diamond, and in particular to make the presentation of a simple and intuitive user interface possible, several restrictions are imposed on RELAX NG schemas used in Spud.

- Choice nodes must be choices between single elements (e.g. a choice between a OR (b AND c) is invalid).
- Elements with the same tag under the same parent are allowed. However, at most one can be + (oneOrMore) or * (zeroOrMore). Elements with the same tag under the same parent must each

have a name attribute with a unique value.

- `name` attributes may contain only alpha-numeric characters, or characters in the set "`/_:[]`"
- Recursive schema elements are not supported.

2.5 Comments and annotations

There are three layers of comment which are applicable in Spud. The schema, as with any piece of source code, can contain comments which are of use to other developers editing the schema. Second, the schema can embed documentation for the problem description language. This documentation will be displayed to the model user by Diamond. The former comments are known as schema comments and are written with a single leading hash (`#`) while the latter are known as schema annotations and are written with a double leading hash (`##`). Schema annotations must be written immediately before the element they document.

Finally, the `comment` named pattern will cause diamond to associate a user comment box with the parent element. This enables users to document their problem description files. Each of the named patterns in the base language also includes the `comment` pattern so that every parameter in an input file can have a user comment associated with it. Figure 2.1 shows a simple schema incorporating all three layers of comment.

2.6 Preprocessing the schema for use with Diamond

RELAX NG comes in two equivalent formats: XML syntax (with file suffix `.rng`) and compact syntax (with file suffix `.rnc`). Compact syntax is optimised for human use, while XML syntax is optimised for ease of machine parsing. Therefore, it is recommended that model developers write the schema in compact syntax, then transform it using a supplied tool to XML syntax for use with Diamond and other validation tools. To transform compact syntax into XML syntax, use the command:

```
spud-preprocess /path/to/schema.rnc
```

This will create a file called schema.rng in the same directory.

Chapter 3

Libspud

Libspud provides C, C++ and Fortran interfaces for accessing the options specified in a Spud XML file.

3.1 The options tree

Spud XML files are read into an in-memory tree structure which reflects the tree of nested elements in the XML. Nodes in this tree are indexed by strings, i.e. the options tree is a dictionary in which values are interrogated via keys.

3.2 Option key syntax

The option key syntax is similar to Unix file path syntax. Consider the simple example, `simple.xml`, which is valid with respect to the schema in figure 2.1:

```
<model_options>
  <simulation_name>
    <string_value lines="1">Basic simulation</string_value>
  </simulation_name>
  <geometry>
    <dimension>
      <integer_value rank="0">3</integer_value>
    </dimension>
  </geometry>
</model_options>
```

The simulation name and the dimension of the geometry may be accessed using the following Fortran program:

```
program fetch_info
  use spud
  implicit none

  integer :: dimension
  character(len=255) :: simulation_name

  call load_options("simple.xml")
  call get_option("/simulation_name", simulation_name)
```

```
  call get_option("/geometry/dimension", dimension)
end program fetch_info
```

The option key `/simulation_name` accesses the element called `simulation_name` that is a child of the root element (which in this case is called `model_options`). The option key `/geometry/dimension` accesses the element `dimension` which in turn is a child of the root element.

3.2.1 Multiple elements

One of the restrictions that Spud places on RELAX NG schemas is that *each element of the same name under the same parent must be differentiated by a `name` attribute*. An example will clarify what is valid. Consider the file `complex_invalid.xml`:

```
<model_options>
  <simulation_name>
    <string_value lines="1">Basic simulation</string_value>
  </simulation_name>
  <geometry>
    <dimension>
      <integer_value rank="0">3</integer_value>
    </dimension>
    <mesh>
      <string_value type="filename" lines="1">mesh_A.msh</string_value>
    </mesh>
    <mesh>
      <string_value type="filename" lines="1">mesh_B.msh</string_value>
    </mesh>
  </geometry>
</model_options>
```

This file is invalid as there are two `mesh` elements beneath the same `geometry` element, and they are not differentiated by a unique `name` attribute. To make this file valid we must differentiate the two `mesh` elements by adding a `name` attribute:

```
<model_options>
  <simulation_name>
    <string_value lines="1">Basic simulation</string_value>
  </simulation_name>
  <geometry>
    <dimension>
      <integer_value rank="0">3</integer_value>
    </dimension>
    <mesh name="PositionMesh">
      <string_value type="filename" lines="1">mesh_A.msh</string_value>
    </mesh>
    <mesh name="VelocityMesh">
      <string_value type="filename" lines="1">mesh_B.msh</string_value>
    </mesh>
  </geometry>
</model_options>
```

This file is now valid as the two `mesh` elements have different `name` values.

To access these elements in the model, the option keys

```
/geometry/mesh[0]
/geometry/mesh[1]
```

may be used to access the elements in order, or they may be accessed by `name` by

```
/geometry/mesh::PositionMesh
/geometry/mesh::VelocityMesh
```

3.2.2 Attributes

For simplicity, attributes of an element are treated the same as children of the element. Consider the example above: `name` is an attribute of the `mesh` element. The `name` of the first `mesh` element may be accessed by

```
/geometry/mesh[0]/name
```

that is, the `name` attribute is accessed the same way as a child element called `name` would be.

Attributes in the options tree must have string type data and have no children. If either of these rules are broken (e.g. via a 3.5.15 call) then the attribute element will be unmarked as an attribute, will be treated as a normal element, and will appear in XML files written out by libspud as XML elements rather than element attributes.

3.2.3 Data elements

Data defined in the Spud base language (see 2.2), such as `integer_value` or `real_value`, are detected by libspud and stored in "`__value`" children. e.g, for the following schema:

```
<model_options>
  <real_parent>
    <ref name="real_value"/>
  </real_parent>
</model_options>
```

the data element "`real_value`" has option key "`/real_parent/__value`". Alternatively, the data element "`real_value`" can be accessed directly with option key "`/real_parent`" - i.e. libspud automatically navigates into "`__value`" child elements when reading or setting options, if such a child element exists.

Manually creating a "`__value`" child for an element that already itself contains data will result in the data of the parent element being removed, and a warning message will be sent to standard error.

3.3 Language specific features

In some cases the interfaces differ slightly between Fortran, C and C++. This is brought about by differences in the designs of these languages themselves and in particular the difference in the manner in which optional arguments are supported in Fortran on the one hand and C/C++ on the other. The decision has been made to write the interfaces in the manner which seems natural in each language at the cost of consistency between languages rather than enforcing a foreign paradigm on one or all of the interfaces.

3.3.1 Fortran

All of the Fortran procedures as well as the named constants for error codes (3.5.1) and data types (3.5.2) are encapsulated in the `spud` module.

Where a routine returns an error code, in Fortran this is achieved via the optional **`stat`** argument. If **`stat`** is not present and an error code other than `SPUD_NO_ERROR` is returned then execution will halt with an error message.

3.3.2 C

Since C does not itself have any namespacing facility, all exposed symbols have the prefix `cspud_`. Error codes are returned via function return values.

3.3.3 C++

The entire public C++ API of libspud is contained in the `Spud` namespace. Error codes are returned via function return values.

3.4 Naming conventions

Where a routine returns its main result via an argument (as is the case for a Fortran subroutine, for example), the routine's name starts with `get_`. The word `key` is exclusively used to refer to a lookup key in the options dictionary.

3.5 Procedure interfaces

In each case, the Fortran interface is given first, followed by the C and then C++ interfaces.

3.5.1 Error codes

The following values are return statuses of procedures. In Fortran these are named constants in the `spud` module while in C and C++ these are the enum types `SpudOptionError` and `Spud::OptionError` respectively.

Error values are greater than zero, warnings are negative and `SPUD_NO_ERROR` has the value 0.

Error code	Interpretation
`SPUD_NO_ERROR`	Successful completion.
`SPUD_KEY_ERROR`	The specified option is not present in the dictionary.
`SPUD_ERROR`	The specified option has a different type from that of the option argument provided.
`SPUD_RANK_ERROR`	The specified option has a different rank from that of the option argument provided.
`SPUD_SHAPE_ERROR`	The specified option has a different shape from that of the option argument provided.
`SPUD_NEW_KEY_WARNING`	The option being inserted is not already in the dictionary.
`SPUD_ATTR_SET_FAILED_WARNING`	The option being set as an attribute can not be set as an attribute.

3.5.2 Data type parameters

The option_type routine returns the following values. In Fortran these are named constants in the `spud` module while in C and C++ these are the enum types `SpudOptionType` and `Spud::OptionType` respectively.

Fortran	C/C++
SPUD_REAL	SPUD_DOUBLE
SPUD_INTEGER	SPUD_INT
SPUD_NONE	SPUD_NONE
SPUD_CHARACTER	SPUD_STRING

3.5.3 clear_options

```
subroutine clear_options()
end subroutine clear_options
```

```
void cspud_clear_options()
```

```
void Spud::clear_options();
```

Clears the entire options tree.

3.5.4 load_options

```
subroutine load_options(filename)
  character(len=*), intent(in) :: filename
```

```
void cload_options(const char* key, const int* key_len)
```

```
void Spud::load_options(const std::string& filename)
```

Reads the XML file `filename` into the options tree.

3.5.5 write_options

```
subroutine write_options(filename)
  character(len=*), intent(in) :: filename
```

```
void cwrite_options(const char* filename, const int* filename_len)
```

```
void write_options(const std::string& filename)
```

Writes the options tree out to the XML file `filename`.

3.5.6 get_child_name

```
subroutine get_child_name(key, index, child_name)
  character(len=*), intent(in)::key
  integer, intent(in)::index
  character(len=*), intent(out)::child_name
```

```
int cspud_get_child_name(const char* key, const int* key_len,
  const int* index,
  char* child_name, const int* child_name_len)
```

```
Spud::OptionError Spud::get_child_name(const std::string& key,
  const unsigned& index,
  std::string& child_name)
```

Retrieves the name of the `index`th child of `key`. This is mostly useful for debugging input files.

Returns error code `SPUD_KEY_ERROR` if the supplied key does not exist in the options tree.

3.5.7 number_of_children

```
function number_of_children(key)
  integer :: number_of_children
  character(len=*), intent(in) :: key
```

```
int cspud_number_of_children(const char* key, const int* key_len)
```

```
int Spud::number_of_children(const std::string& key)
```

Returns the number of children under `key`. This is mainly of use for debugging input files.

Returns 0 if the specified key does not exist in the options tree.

3.5.8 option_count

```
function option_count(key)
  integer :: option_count
  character(len=*), intent(in) :: key
```

```
int cspud_option_count(const char* key, const int* key_len)
```

```
int Spud::option_count(const std::string& key)
```

Returns the number of options which match `key`. Searches all possible paths matching the given key. For example, for the following XML file:

```
<model_options>
  <scalar_field name="pressure">
    <solver>
      ...
    </solver>
  </scalar_field>
  <scalar_field name="temperature">
    <solver>
      ...
    </solver>
  </scalar_field>
</model_options>
```

in the following Fortran code:

```
n_child = option_count("/scalar_field/solver")
```

`n_child` is assigned the value 2.

This routine is useful where an option can occur any number of times (for example a simulation may allow for any number of fields to be specified).

Returns 0 if `key` is not present in the dictionary.

3.5.9 have_option

```
function have_option(key)
  logical :: have_option
  character(len=*), intent(in) :: key
```

```
int cspud_have_option(const char* key, const int* key_len)
```

```
logical_t Spud::have_option(const std::string& key)
```

Returns true if `key` is present in the options dictionary, and false otherwise. This is useful for determining whether optional options have been set and for determining which of a choice of options has been selected.

3.5.10 option_type

```
function option_type(key, stat) result (type)
  integer :: type
  character(len=*), intent(in) :: key
  integer, optional, intent(out) :: stat
```

```
int cspud_get_option_type(const char* key, const int* key_len, int* type)
```

```
Spud::OptionError Spud::get_option_type(const std::string& key,
Spud::OptionType& type)
```

Returns the type of the option specified by `key`. The type will be returned as one of the named constants in 3.5.2.

Returns error code `SPUD_KEY_ERROR` if the supplied key does not exist in the options tree.

3.5.11 option_rank

```
function option_rank(key, stat) result (rank)
  integer :: rank
  character(len=*), intent(in) :: key
  integer, optional, intent(out) :: stat
```

```
int cspud_get_option_rank(const char* key, const int* key_len, int* rank)
```

```
Spud::OptionError Spud::get_option_rank(const std::string& key,
int& rank)
```

Returns the rank of the option specified by `key`. The rank returned will be 0 (for a scalar), 1 (for a vector) or 2 (a rank 2 tensor, or matrix).

Returns error code `SPUD_KEY_ERROR` if the supplied key does not exist in the options tree.

3.5.12 option_shape

```
function option_shape(key, stat) result (lshape)
  integer, dimension(2) :: lshape
  character(len=*), intent(in) :: key
  integer, optional, intent(out) :: stat
```

```
int cspud_get_option_shape(const char* key, const int* key_len, int* shape)
```

```
Spud::OptionError Spud::get_option_shape(const std::string& key,
std::vector<int>& shape)
```

Returns the shape of the option specified by `key`. The shape is always a 2-vector. If the option in question is rank 1 then the second component will be -1, if the option is rank 0 (a scalar) then both entries will be -1.

Returns error code `SPUD_KEY_ERROR` if the supplied key does not exist in the options tree.

3.5.13 get_option

```
subroutine get_option(key, val, stat, default)
  character(len=*), intent(in) :: key
  option_type, intent(out) :: val
  integer, optional, intent(out) :: stat
  option_type, optional, intent(in) :: default
```

```
int cspud_get_option(const char* key, const int* key_len, void* val)
```

```
Spud::OptionError Spud::get_option(const std::string& key,
option_type val)

Spud::OptionError Spud::get_option(const std::string& key,
option_type val, option_type default_val)
```

This is the main method for retrieving option values from the options dictionary. For Fortran and C++ *option_type* can be any of the following values:

Fortran type	C++ type
`double precision`	`double&`
`double precision, dimension(:)`	`std::vector<double>&`
`double precision, dimension(:,:)`	`std::vector< std::vector<double> >&`
`integer`	`int&`
`integer, dimension(:)`	`std::vector<int>&`
`integer, dimension(:,:)`	`std::vector< std::vector<int> >&`
`character(len=*)`	`std::string&`

In Fortran, single precision interfaces are also provided in each of the above cases. However, these values will be stored in the options dictionary in double precision. In every case the type and shape of the argument must match that of the option in the dictionary. In C the returned argument is always a `void` pointer and it is the responsibility of the user to match the size and type correctly.

This routine can return the following error codes:

- If `key` matches an option but the shape, rank or type fails to match, appropriate error code will be set (see 3.5.1).
- If `key` fails to match but `default` is present, `option` is set to the value of `default`.
- If `key` fails to match and `default` is not present, the error code will be set to `SPUD_KEY_ERROR`.

3.5.14 add_option

```
subroutine add_option(key, stat)
  character(len=*), intent(in) :: key
  integer, optional, intent(out) :: stat
```

```
int cspud_add_option(const char* key, const int* key_len)
```

```
Spud::OptionError Spud::add_option(const std::string& key)
```

Creates a new option at the supplied key. If the option does not currently exist, creates the option (with data type `SPUD_NONE`) and returns error code `SPUD_NEW_KEY_WARNING`.

3.5.15 set_option

```
subroutine set_option(key, val, stat)
  character(len=*), intent(in) :: key
  option_type, intent(in or inout) :: val
  integer, optional, intent(out) :: stat
```

```
int cspud_set_option(const char* key, const int* key_len, const void* val,
const int* type, const int* rank, const int* shape)
```

```
Spud::OptionError Spud::set_option(const std::string& key,
const option_type& val)
```

Method for setting options in the options tree. The *option_type* can be any of the types listed above in 3.5.13.

This routine can return the following error codes:

- If `key` matches an option but the shape, rank or type fails to match currently existing option, returns an appropriate error code (see 3.5.1).
- If `key` fails to match, creates a new option at the supplied key, sets the option to `val` and returns error code `SPUD_NEW_KEY_WARNING`.

3.5.16 set_option_attribute

```
subroutine set_option_attribute(key, val, stat)
  character(len=*), intent(in) :: key
  character(len=*), intent(in) :: val
  integer, optional, intent(out) :: stat
```

```
int int cspud_set_option_attribute(const char* key, const int* key_len,
const char* val, const int* val_len)
```

```
Spud::OptionError Spud::set_option_attribute(const std::string& key,
const std::string& val)
```

As `set_option` (see 3.5.15), but additionally attempts to mark the option at the specified key as an attribute. Note that `set_option_attribute` accepts only string data data for `val`.

This routine can return the following error codes:

- If `key` matches an option but the shape, rank or type fails to match currently existing option, returns an appropriate error code (see 3.5.1).
- If `key` fails to match, creates a new option at the supplied key, sets the option to `val` and returns error code `SPUD_NEW_KEY_WARNING`.
- If `key` matches an option, but the existing option has children, sets the option to `val` and returns error code `SPUD_ATTR_SET_FAILED_WARNING`.

3.5.17 delete_option

```
subroutine delete_option(key, stat)
  integer, optional, intent(out) :: stat
```

```
int cspud_delete_option(const char* key, const int* key_len)
```

```
Spud::OptionError Spud::delete_option(const std::string& key)
```

Deletes the option at the specified key.

Returns error code `SPUD_KEY_ERROR` if the supplied key does not exist in the options tree.

3.5.18 print_options

```
subroutine print_options()
```

```
void cspud_print_options()
```

```
void Spud::print_options()
```

Prints the entire options tree to standard output. Useful for debugging.

Chapter 4

Diamond

4.1 Installation

Diamond depends on Python (version 2.3 or greater), PyGTK, the 4Suite XML library, and the lxml library. All of these dependencies are available in the Debian/Ubuntu repositories with the following command:

```
apt-get install python-gtk2 python-lxml python-4suite-xml
```

Windows versions of these packages are available for download on their respective websites.

4.2 Configuration files

In accordance with normal Unix practice, system wide configuration for diamond is stored in the `/etc/diamond` directory while per-user configuration is stored in a `.diamond` directory in the user's home directory.

4.2.1 Schemas

Diamond needs to know which Spud schemas are installed and available on the current system. This is specified in the `schemata` subdirectory of the `/etc/diamond` and `~/.diamond` directories. The file names in the `schemata` directory give the filename extension associated with a particular problem description language. The content of the file is two lines. The first line is the name of the problem description language while the second line contains the path of the XML syntax (`.rng`) version of the corresponding schema.

For example, the Fluidity package has a problem description language called the Fluidity Markup Language which uses the file extension `.flml`. When installed on a system by the sysadmin, Fluidity might create the file `/etc/diamond/flml` with the following contents:

```
Fluidity Markup Language
/usr/share/fluidity/fluidity_options.rng
```

An individual user `jrluser` might have the current version of the Fluidity svn tree checked out in their home directory and would need to point diamond at the (posssibly updated) schema in their source tree. `jrluser` would then create the file `/home/jrluser/.diamond/schemata/flml` which would contain:

```
Fluidity Markup Language
/home/jrluser/fluidity/tools/fluidity_options.rng
```

Now Diamond will pick up the version of the schema in `jrluser`'s svn tree rather than the version the sysadmin installed.

It is also possible to specify a URL over HTTP for the location of the schema. For example:

```
Fluidity Markup Language
http://amcg.ese.ic.ac.uk/svn/fluidity/trunk/tools/fluidity_options.rng
```

This is to facilitate centralised deployments.

4.2.2 Running Diamond

Diamond can be started in several ways.

When executed as

```
diamond
```

Diamond will offer the user the choice of which registered schema to use. If only one schema is registered, then it will use that by default. Diamond will open a blank document of the language specified by the schema to be edited.

When executed as

```
diamond -s /path/to/schema.rng
```

Diamond will open a blank document using the schema specified on the command line.

When executed as

```
diamond filename.suffix
```

Diamond will inspect the registered schemas for the suffix specified and use that schema.

4.2.3 Dynamic validation

Diamond uses the schema to guide the editing of valid documents that the schema permits. When a blank document is opened, information necessary to make a valid document is missing and the document is thus invalid[1]. The invalidity of the document is reflected in the colouration of the root node: valid nodes are coloured in black, while invalid nodes are coloured in blue. As invalid nodes have their attributes and data specified, their validity is dynamically updated and they are coloured appropriately. The document as a whole is valid if and only if the root node is coloured black.

[1] This is true for all but a trivial schema which permits only one document.

Chapter 5

Miscellaneous tools

5.1 Spud-set

Spud-set is a short python script which enables users to make small modifications to the values of options in Spud options files on the command line. This is particularly useful where users wish to script a number of model runs exploring different values of one or more parameters.

The syntax of spud-set is:

```
spud-set filename xpath new_value
```

where:

filename is the name of the options file which is to be modified.

xpath is the xpath of the option to be modified (see below).

new_value is the new value which the option should take.

5.1.1 xpath

An xpath is a language which provides a mechanism for addressing parts of an xml file. It uses paths similar in some ways to those used to retrieve options in libspud. A full description of xpath syntax is to be found in the World Wide Web Consortium XPath Recommendation. However, users of Spud will usually find it much easier to use the xpaths generated by Diamond. The xpath for any option is shown at the bottom of the Diamond window when that option is selected and can be copied to the clipboard using the "copy spud path" option from the edit menu.